To Ponder

Evaluate: True or false?

\[
\begin{align*}
true & \equiv '1' \\
'false' & \equiv false \\
0 & \equiv '0' \\
0 & \equiv '' \\
NaN & \equiv NaN
\end{align*}
\]
JavaScript: Introduction, Types, Functions

Lecture 15
History

- Developed by Netscape
  - “LiveScript”, then renamed JavaScript
  - *Nothing* to do with Java!

- Motivation: client-side execution in browser
  - Interpretted

- Standardized by ECMA (“ECMAScript”)
  - Big update v6 in 2015, ie ES6 (aka ES15)
  - Now annual updates, every June
  - After ES6, named with year (eg ES20)

- Has become popular outside of browsers
  - *e.g.* Node.js

- Translation target for other languages:
  - Syntax: CoffeeScript
  - Static types: Dart (Google), TypeScript (MS)
Client-Side Execution

GET /news/index.php HTTP/1.1
Host: www.osu.edu
User-Agent: Mozilla/5.0 (X11; Ubuntu;...etc

<!DOCTYPE html>
<html lang="en">
  <head><title>My Page</title>
  <meta charset="utf-8" />
  ...
</html>
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Something Short and Sweet</title>
        <meta charset="utf-8" />
    </head>
    <body>
        <p>Hello <a href="planet.html">World</a>!</p>
        <br />
        <img src="globe.png" alt="a globe"/>
    </body>
</html>
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Something Short and Sweet</title>
    <meta charset="utf-8" />
    <script>
        window.alert("Annoying!");
    </script>
</head>
<body>
    <p>
        Hello <a href="planet.html">World</a>!
    </p>
    <img src="globe.png" alt="a globe"/>
</body>
</html>
Including Scripts

- Head: executed before body displays
  - Script (source) can be explicitly included
    
```html
<script>
    console.info("hi");

    ...
</script>
```
  - Script can be linked in from external file
    
```html
<script src="MyProgram.js"></script>
```
  - Recall: linking to CSS

- Inline: executed as body is displayed

- Browser blocks while downloading
  - Common advice: put scripts at end of body
  - Modern advice: use `<script src="..." async>`
Async/defer Downloading

defer
async

parser  fetch  execution
Demo

- Simple “hello world” (page1.html)
  - HTML file containing JavaScript
  - Body is empty, script writes HTML output
  - Browser displays result

- Examining result with dev tools
  - Sources: see JavaScript program
    - Place breakpoints and reload
  - Console: see console output
Some Objects Provided Implicitly

- Some objects are created implicitly by the execution environment (browser)
- Document object (document)
  - document.writeln() puts output in body
- Window object (window)
  - Refers to browser's display window
  - Alert method pops up a dialogue
    ```javascript
    window.alert("Say \"cheese\"!");
    ```
  - Prompt method pops up a dialogue
    ```javascript
    name = window.prompt("Enter name");
    ```
Demo with Popups

- See: codepen.io/cse3901/pen/BYqqPb
  - Alert window
  - Prompt window
  - Console output (info, warn, error)

- Notice:
  - HTML body is empty
  - Settings > Auto-update preview (Off)
Familiar (Java) Minor Syntax

- Statement separator ;
  - Wrinkle: ;'s are optional!
    - Implicitly automatically inserted
    - But clearer and safer to include explicitly
- Statement blocks {...}
- Parentheses in expressions (…)
- Comments // and /*…*/
Familiar (Java) Operators

- Arithmetic (numbers are floats)
  - `+` - `*` - `/` - `%`
  - Wrinkles:
    - No diff in `/` between ints and floats!
    - `%` works on floats!

- Relational
  - `<` - `>` - `<=` - `>=`
  - `==` - `!=`
  - Wrinkle: `===` - `!==`

- Logical
  - `&` - `|` - `!`
Familiar (Java) Statements

- **Assignment**
  - `=`
  - `+= -= *= /= %=`
  - `++ --` (pre and post)

- **Conditionals**
  - `if (...)`, `if (...) ... else`
  - `switch (c)`
    - `case 'a': ... case 'b': ... default;`

- **Iteration**
  - `while (...)`, `do...while(...)`
  - `for (...;...;...)`
  - `break, continue`
Primitive vs Reference Types

- Distinction is similar to Java
- A variable is a “slot” in memory
- A variable can be *primitive*
  - The slot holds the value itself
  - Boolean, number, string, (null, undefined)
  - Since ECMAScript 2015 (ES6): symbols
- A variable can be a *reference*
  - The slot holds a pointer to the value
  - Arrays and objects (including functions!)
Primitive vs Reference Types

- \(a\): 34.2
- \(b\): "hi"
- \(c\): 4
- \(d\): width: 12, height: 15, color: "blue"
Primitives: Checking Equality

```javascript
let a = 5;
let b = 5;
let c = 7;

if (a == b) ... //=> true, equal slots
if (a == c) ... //=> false

let x = "hello";
let y = "hello";

if (x == y) ... //=> true! cf. Java
```
Primitives: Assignment is Copy

```javascript
let a = 5;
let b = a;  // copy contents of slot

b++;

if (a == 5)…  //=> true, a unchanged
```
Assignment is Copy (of Slot)

```javascript
let a = 5;
let b = a;
b++;
if (a == 5)...
```
Primitives: Argument Passing

```javascript
function inc (param) {
    param++;
}

let a = 5;
inc(a);  // copy contents of slot
if (a == 5) ...  //=> true
```
let a = {x:1, y:4}; // a new object
let b = {x:1, y:4}; // a new object

if (a == b) ... //=> false

a = b; // copy contents of slot

if (a == b) ... //=> true
Assignment is Copy (of Slot)

\[ a = b; \]

\[ a \neq b \]

\[ a == b \]
References: Argument Passing

```javascript
function inc (param) {
    param.x++; 
}

let a = {x: 1, y: 4};
inc(a); // copy contents of slot
if (a.x == 2)...  //=> true
```
function inc (param) {
    param = {x: 2, y: 7};
}

let a = {x: 1, y: 4};
inc(a); // copy contents of slot
if (a.x == 2) //=> false
Wrinkle: `==` vs `===`

- Recall `+` operator in Java
  - Concatenation between strings
  - Addition between numbers
  - `3 + "4"` also works! Results in "34"

- Similarly, JavaScript `== (!=)` tries to make types match
  - `3 == "3"` is true!

- To prevent implicit type conversion, use `=== (!==)`
  - `3 === "3"` is false

- More on type conversion later...
Demo: Iteration

- See: codepen.io/cse3901/pen/Jpmejp
- Table generated by Javascript
  - Prompt for initial value
  - Calculate interest series
  - Print out a row of table for each year
Static vs Dynamic Types

- **Static:** known at compile time
  - *e.g.*, C, C++, Java, Ada
    - `int x`
    - `char[] a`
    - `FluffyCloud t`
    - `void* d`

- **Dynamic:** known only at run time
  - *e.g.*, Python, PHP, Ruby, JavaScript
    - `let x`
    - `let a`
    - `let t`
    - `let d`
Static Types

- **a**: 34.2 (number)
- **b**: "hi" (string)
- **c**: num[] (two arrays: 4, 0, -300, 3.14)
- **d**: Shape (width: 12, height: 15, color: "blue")
Dynamic Types

- \( a \): 34.2
- \( b \): "hi"
- \( c \):
  - let
  - 4
  - 0
  - -300
  - 3.14
- \( d \):
  - let
  - width: 12
  - height: 15
  - color: "blue"
  - Object

\[ \text{let [ ]} \]
Function Signatures

- Statically typed
  
  ```java
  String parse(char[] s, int i) {... return e;}
  out = parse(t, x);
  ```

  - Parameter types (i.e. s and i) are declared
  - Return type (i.e. of parse) is declared
  - The compiler checks conformance of
    - (Declared) types of arguments (t, x)
    - (Declared) type of return expression (e)
    - (Declared) type of expression using parse (out)

- Dynamically typed
  
  ```javascript
  function parse(s, i) { ... }
  out = parse(t, x)
  ```

  - You are on your own!
Changing Types at Run-time

Static Types

// a is undefined
String a;

// a is null string
a = "hi;"

// compile-time err
a = "hi";

a = 3;

// compile-time err
a.push();

Dynamic Types

// a is undeclared
let a;

// a is undefined
a = "hi;"

// load-time error
a = "hi";

a = 3;

// a is a number
a.push();

// run-time error
Resources

- MDN (Mozilla Developer Network)
  - developer.mozilla.org/docs/JavaScript
- codepen.io, jsfiddle.net
  - HTML, CSS, Javascript → result
- REPL
  - In VM, at console:
    - $ node
  - In a browser: repl.it/languages/javascript
- Class web site (under Resources)
  - Style guides (Airbnb, Google)
  - Books, available online
    - JavaScript: The Definitive Guide (Flanagan)
    - Eloquent JavaScript (Haverbeke)
## Conversion of Primitive Values

<table>
<thead>
<tr>
<th>numbers</th>
<th>string</th>
<th>number</th>
<th>boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;0&quot;</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>&quot;0&quot;</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>&quot;1&quot;</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>&quot;NaN&quot;</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>Infinity</td>
<td>&quot;Infinity&quot;</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>-Infinity</td>
<td>&quot;-Infinity&quot;</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>6.022e23</td>
<td>&quot;6.022e+24&quot;</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>
## Conversion of Primitive Values

<table>
<thead>
<tr>
<th>boolean</th>
<th>string</th>
<th>number</th>
<th>boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>true</code></td>
<td>&quot;true&quot;</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><code>false</code></td>
<td>&quot;false&quot;</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>strings</th>
<th>string</th>
<th>number</th>
<th>boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;&quot;</td>
<td>0</td>
<td></td>
<td>false</td>
</tr>
<tr>
<td>&quot;   &quot;</td>
<td>0</td>
<td></td>
<td>true</td>
</tr>
<tr>
<td>&quot;1.2&quot;</td>
<td>1.2</td>
<td></td>
<td>true</td>
</tr>
<tr>
<td>&quot;0&quot;</td>
<td>0</td>
<td></td>
<td>true</td>
</tr>
<tr>
<td>&quot;one&quot;</td>
<td>NaN</td>
<td></td>
<td>true</td>
</tr>
</tbody>
</table>
## Conversion of Primitive Values

<table>
<thead>
<tr>
<th></th>
<th>string</th>
<th>number</th>
<th>boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>undefined</strong></td>
<td>undefined</td>
<td>&quot;undefined&quot;</td>
<td>NaN</td>
</tr>
<tr>
<td><strong>null</strong></td>
<td>null</td>
<td>&quot;null&quot;</td>
<td>0</td>
</tr>
</tbody>
</table>
Summary of (Simple?) Rules

- How do numbers convert to things?
  - Boolean: 0 is false, non-0 is true (exception: NaN)

- How do strings convert to things?
  - Numbers: non-valid syntax give NaN (exception: empty/blank give 0)
  - Boolean: true, only empty string is false

- How does undefined convert to things?
  - Number: NaN

- How does null convert to things?
  - Number: 0
How do things convert to boolean?
- Empty string is `false`
- Numbers (+/-) 0 and `NaN` are `false`
- `undefined` and `null` are `false`

Aka “falsy” (vs. “truthy”)

Importance: Boolean contexts

```
if (pet) ...
```

Pitfall: `&&`, `||` may not result in a boolean
- `x || y` means `x ? x : y` (first `x` converted)
  
```
p = "cat" || "dog"  //=> p == "cat"
```
- Old idiom: `!!x` forces conversion to boolean
  
```
p = !!("cat" || "dog")  //=> p == true
```
Easier? Column-Major View

- How do things convert to Numbers?
  - Empty (and whitespace) string is 0
  - Non-numeric strings are NaN
  - `undefined` is NaN
  - `null` is 0

- Importance: Used in `==` evaluation
== Evaluation is... Different

- When types do not match, coerce:
  - `null & undefined` (only) equal each other
  - Strings & booleans converted to `numbers`:
    - "1.0" == true && "" == false
  - Pitfall: `NaN` is not equal to `NaN`

- When one operand is an object:
  - Convert via `valueOf` (or `toString`)
  - Result then compared with usual `==` rules
  - Note: no coercion when both operands are references (`==` is reference equality)

- Note:
  - `===` never coerces
To Ponder

Evaluate: True or false?

\[ true == '1' \]

\[ 'false' == false \]

\[ 0 == '0' \]

\[ 0 == '' \]

\[ NaN == NaN \]
Surprising Consequences

false == 'false'  //=>
false == '0'     //=>
!!'0'            //=>
('0' == 0) && (0 == '') && ('0' != '') //=>
(NaN == true) || (NaN == false) //=>
!!NaN             //=>
(NaN != 0) && (!!NaN == !!0) //=>

- dorey.github.io/JavaScript-Equality-Table
Surprising Consequences

false == 'false'  //=> false
false == '0'     //=> true
!!'0'            //=> true
('0' == 0) && (0 == '') &&
    ('0' != '')   //=> true
(NaN == true) || (NaN == false)
    //=> false
!!NaN             //=> false
(NaN != 0) && (!!NaN == !!0)
    //=> true

- dorey.github.io/JavaScript-Equality-Table
Functions are People too

- Named functions: declaration & use
  ```javascript
  function foo(a, b) { ... }
  foo("hi", 3);
  ```

- Anonymous functions
  ```javascript
  function(a, b) { ... }
  // how do we invoke such a thing?
  ```

- Functions are objects (first-class citizens)

- They can be assigned to variables!
  ```javascript
  let foo = function(a, b) {...};
  foo("hi", 3);
  let bar = foo;    // cf. let bar = foo();
  bar("world", 17);
  ```
Functions are Objects

```javascript
Circle

this.centerX = x;
this.centerY = y;
... Etc ...

return Math.PI * 
this.radius * 
this.radius
```
Functions Can Be Arguments

```javascript
function apply(x, a) {
    return x(a); // x is a function!
}

function square(i) {
    return i * i;
}

apply(square, 5) //=> 25
```
Functions Can Be Return Values

```javascript
function grantDegree() {
    function addTitle(name) {
        return "Dr. " + name;
    }
    return addTitle; // a function!
}

let phd = grantDegree();
phd("Turing"); // phd is a function
phd(3/2);    //=> "Dr. 1.5"
```
Closures

```javascript
function greaterThan(bound) {
    function compare (value) {
        return value > bound;
    }
    return compare; // 1-arg function
}

let testPos = greaterThan(0);
testPos(4)    //=> true
testPos(-3)   //=> false
```
Closures + Anonymity

```javascript
function greaterThan(bound) {
    function compare (value) {
        return value > bound;
    }
    return compare;  // 1-arg function
}

let testPos = greaterThan(0);
testPos(4)  //=> true
testPos(-3)  //=> false
```
function greaterThan(bound) {
    let compare = function(value) {
        return value > bound;
    }
    return compare; // 1-arg function
}

let testPos = greaterThan(0);
testPos(4)  //=> true
testPos(-3) //=> false
Closures + Anonymity

```javascript
function greaterThan(bound) {
    return function(value) {
        return value > bound;
    }
}

let testPos = greaterThan(0);

console.log(testPos(4)); // => true
console.log(testPos(-3)); // => false
```
Arrow Function Expressions

- Concise notation for anon. functions
- Syntax:
  - Omit `function` keyword
  - Place arrow `=>` between params and body
  - `(a, b = 10) => { ... }`
  - `(r) => { return Math.PI*r**2 }`
- For one-liner, can omit `return` and `{}`’s
  - `(r) => Math.PI * r**2`
- For one parameter, can omit `()`
  - `r => Math.PI * r**2`
- Use where function expressions needed
  - `let area = r => Math.PI * r**2`
Closures + Anonymity Revisited

```javascript
function greaterThan(bound) {
  return value => value > bound;
}

let testPos = greaterThan(0);

testPos(4)       //=> true

testPos(-3)      //=> false
```
IIFE

- Immediately Invoked Function Expression
  - Define *and* invoke function at the same time

- Basic forms:
  - `(function() { /* code here */ })();`
  - `let n = function() { /* code here */ }();`

- Work-around for weird JavaScript scoping
  - `var` scopes variables to the enclosing *function*
  - IIFE creates a lexical scope (with closures)

- Modern JavaScript has `let` (and `const`)
  - These scope variables to the enclosing *block*
  - General advice: prefer `let` to `var`
  - IIFEs are still encountered in the wild
Summary

- Truthy, falsey, and friends
  - Type coercion is everywhere
  - Coerce to boolean in conditionals
  - Coerce to number for ==

- Functions as first-class citizens
  - Can be passed as arguments
  - Can be returned as return values!
  - Closure: carry their context